Squiggle
Modelling SQL with associated type synonyms

Ganesh Sittampalam
Credit Suisse
ganesh.sittampalam@credit-suisse.com

This talk is a lie

http://code.haskell.org/squiggle/unstable
has the gory truth

Including liberal use of
`unsafeCoerce`

Squiggle is nowhere near finished

What is Squiggle?

- A type-safe embedding of SQL in Haskell
- DSELs rule!

Why embed SQL?

- Type safety
- Compositionality
  - Optimisation (abstraction removal)

Why not HaskellDB?

- Want to write natural Haskell code
- Simpler types
  - Decoupled from fieldnames
- Avoid the artificial monad
- Overloading between Haskell-time and SQL-time

It's a large design space

Not trying to model relational algebra

Associated type synonyms

```
type family Nullable a

type instance Nullable Int = Maybe Int
(…)

type instance Nullable (Maybe a) = Maybe a

type instance Nullable () = ()
type instance Nullable (a, b)
  = (Nullable a, Nullable b)
type instance Nullable (a, b, c)
  = (Nullable a, Nullable b, Nullable c)
```

A football match

```
data Result = Result {
    r_date :: Date,
    r_home :: String,
    r_away :: String,
    r_homegoals :: Int,
    r_awaygoals :: Int
}
```

Apply

```
newtype Id a = Id a
newtype Comp f g x = Comp (f (g x))

type family Apply (f :: * -> *) a

type instance Apply Id a = a
type instance Apply (Comp f g) a
  = Apply f (Apply g a)
type instance Apply SqlExpr a
  = SqlExpr a
```

A football match (overloaded)

```
data Result s = Result {
    r_date :: Apply s Date,
    r_home :: Apply s String,
    r_away :: Apply s String,
    r_homegoals :: Apply s Int,
    r_awaygoals :: Apply s Int
}
```

s ∈ { Id, SqlExpr }

Calculating stuff

```
points :: Result s
       -> (Apply s Int,
           Apply s Int)

points Result{..}
 = r_homegoals > r_awaygoals ?
     ((3, 0),
      r_homegoals == r_awaygoals ?
      ((1, 1),
       (0, 3)))
```

So what about some SQL?

```
results :: Query (Result SqlExpr)
results =
  table (Nothing, "results") $
  Result {
   r_date = field "date",
   r_home = field "home",
   r_away = field "away",
   r_homegoals = field "homegoals",
   r_awaygoals = field "awaygoals“
  }
```

Now what?

```
sortedResults :: Query (Result SqlExpr)
sortedResults =
  sortOn r_date results
```

```
sortOn :: (a -> b) -> Query a -> Query a
```

On lists this would be

```
sortOn :: (a -> b) -> [a] -> [a]
sortOn f = sortBy (compare `on` f)
```

More combinators (1)

```
map :: (a -> b)
    -> Query a -> Query b

filter :: (a -> SqlExpr Bool)
       -> Query a -> Query a

cross :: Query a -> Query b
      -> Query (a, b)
```

More combinators (2)

```
fromList :: [a] -> Query a

union :: Query a -> Query a
      -> Query a

cross :: Query a -> Query b
      -> Query (a, b)

nub :: Query a -> Query a
```

Grouping

```
groupOn :: (a -> g) -> Query a
        -> (Aggr a -> b) -> Query b
```

Perhaps it should be

```
groupOn :: (a -> g) -> Query a
        -> Query (Aggr a)
```

Getting the results

```
run :: HasSqlExpr a sea
    => Connection -> Query sea -> IO [a]

HasSqlExpr (Result Id) (Result SqlExpr)
(…)
```

Overloading syntax

- Num ☺
- -XOverloadedStrings ☺

- Booleans ☹
- Lists, list comprehensions ☹
- If-then-else ☹
- Pattern matching ☹

- Not relevant to this talk: do-notation 😐

Overloading and associated type synonyms

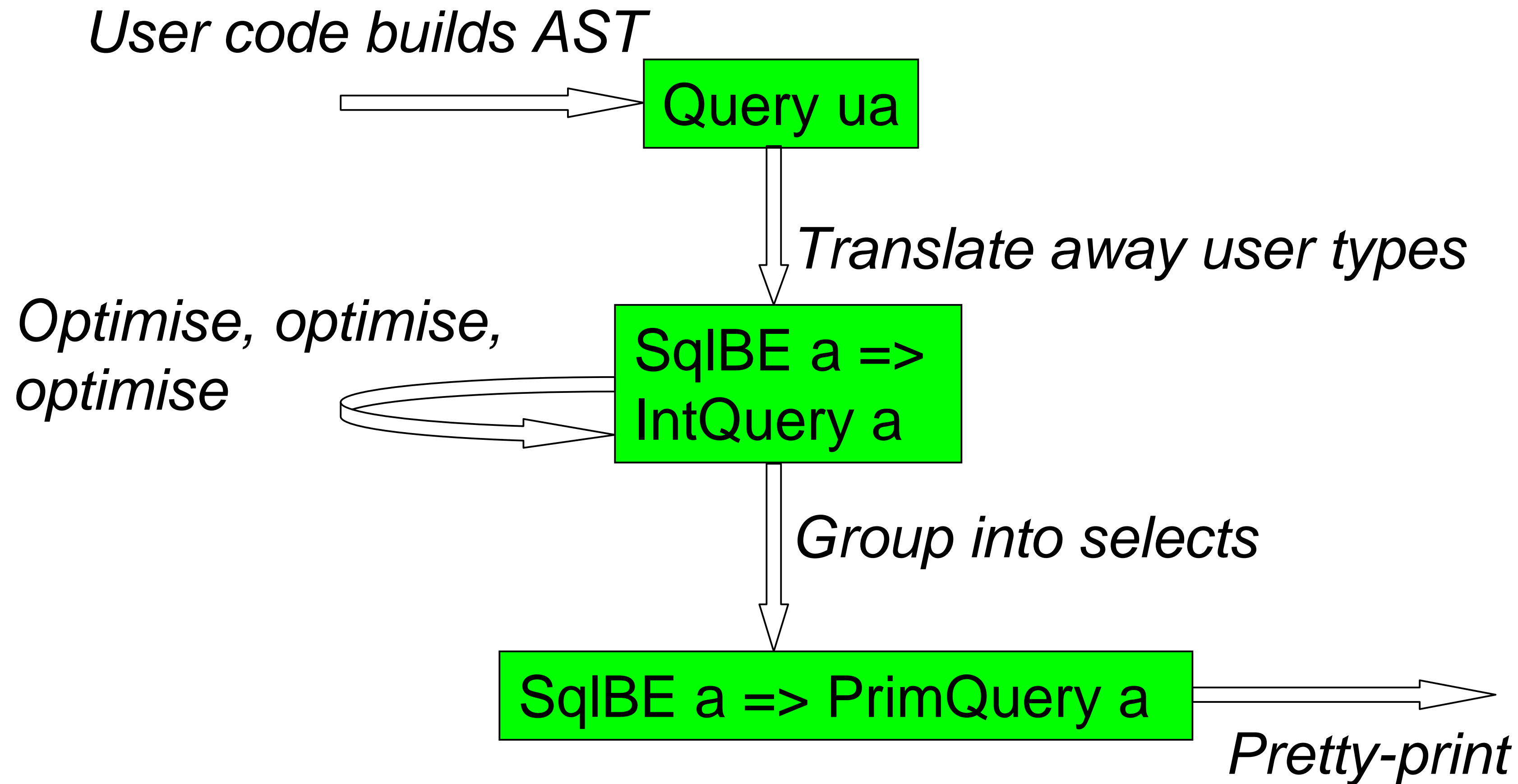
```
type family Id a
type instance Id Int = Int

foo :: Id a -> Id a
foo = id

foo2 :: Id a -> Id a
foo2 = foo
```

```
 Couldn't match expected type `Id a1' against
   inferred type `Id a'
     In the expression: foo
     In the definition of `foo2': foo2 = foo
```

Encoding

- Need a uniform representation
- Transform user types into nested pairs
  - Type class defines the transformation
  - Template Haskell to auto-generate it

Query compiler



Type-safe right until we generate text

Higher-order abstract syntax

- Two ways to represent functions:
  a) Explicit representation of lambda in AST
  b) Implicit, i.e. reuse Haskell's function type

a) harder to make type-safe
   name hygiene problems

b) somewhat harder to optimise
   hard to manipulate efficiently

Intermediate queries

```
data IntQuery a where
   IntKnown :: SqlBE a
            => [SqlExpr a] -> IntQuery a
   IntTable :: SqlBE a
            => (Maybe DBName, TableName)
            -> SqlFields a -> IntQuery a
   IntProject :: (SqlBE a, SqlBE b)
              => (SqlExpr a -> SqlExpr b)
              -> IntQuery a -> IntQuery b
   IntRestrict :: SqlBE a
               => (SqlExpr a -> SqlExpr Bool)
               -> IntQuery a -> IntQuery a
   (…)
```

SQL expressions

```
data SqlExpr' q a where
  SEField :: SqlPrimType a
          => (Bool, Maybe TableName)
          -> FieldName
          -> SqlExpr' q a
  SEPrim :: SqlPrim a -> SqlExpr' q a
  SEApp :: SqlExpr' q (a -> b)
        -> SqlExpr' q a
        -> SqlExpr' q b
  (…)
```

SQL expressions

```
data SqlExpr' q a where
  (…)
  SEAggr :: SqlExpr' q a
         -> SqlExpr' q (Aggr a)
  SEProd :: SqlExpr' q a -> SqlExpr' q b
         -> SqlExpr' q (Prod a b)
  SENil :: SqlExpr' q Nil
  (…)
```

SQL expressions

```
data SqlExpr' q a where
   (…)
   SEIntQuery :: SqlBE a
              => IntQuery a
              -> SqlExpr' IntQuery [a]
   SEPrimQuery :: SqlBE a
               => PrimQuery a
               -> SqlExpr' PrimQuery [a]

type SqlExpr a = SqlExpr' IntQuery a
```

SQL primitives

```
data SqlPrim a where
  SPLit  :: SqlLit a -> SqlPrim a
  SPPlus :: (SqlPrimType a, Num a)
         => SqlPrim (a -> a -> a)
  (…)
```

Optimising

```
optSE :: SqlBE a => SqlExpr a -> SqlExpr a
optSE (SEApp (SEPrim SPNot) e)
   | SEPrim (SPLit (SLBool b)) <- e'
      = SEPrim (SPLit (SLBool (not b)))
   | SEApp (SEPrim SPNot) ne <- e' = ne
 where e' = optSE e
(…)
optSE (SEApp f a) = SEApp (optSE f) (optSE a)
(…)
optSE e = e
```

Facts

```
data Fact q where
  Fact :: SqlExpr a -> SqlExpr a -> Fact

class Same s where
  sameT :: s a -> s a' -> Maybe (EqTypes a a')

data EqTypes a a' where
  EqId :: EqTypes a a

applyFact :: Fact -> SqlExpr x -> SqlExpr x
applyFact (Fact lhs rhs) e
  = case sameT lhs e of
      Just EqId -> rhs
      Nothing -> e
```

Dead code removal

```
removeDead :: SqlBE a
           => IntQuery a -> IntQuery a
removeDead = removeDead' id

removeDead' :: (SqlBE a, SqlBE b)
            => (SqlExpr a -> SqlExpr b)
            -> IntQuery a -> IntQuery b
…
removeDead' f q = IntProject f q
```

Dead code removal

```
decompose :: (SqlBE a, SqlBE c)
          => (SqlExpr a -> SqlExpr c)
          -> exists b .
             (SqlBE b =>
               (SqlExpr a -> SqlExpr b,
                SqlExpr b -> SqlExpr c))
```

Dead code removal

```
removeDead' f (IntProject g q)
 = case decompose (f . g) of
      (f', g')
         -> IntProject g' (removeDead f' q)
```

Rebuilding

```
optSEFunc :: (SqlBE a, SqlBE b)
          => (SqlExpr a -> SqlExpr b)
          -> (SqlExpr a -> SqlExpr b)
optSEFunc f = optSE . F
```

- Inefficient
- Breaks when combined with dead-code removal

- We need a way to collapse out the internal computations
  - `rebuildFunc` (same type as `optSEFunc`) does this

That's all folks

http://code.haskell.org/squiggle/unstable